下载APP

12 | 反爬虫概述（四）：前后端都不合适的时候如何进行处理？

2022-02-25 DS Hunter

《反爬虫兵法演绎20讲》 课程介绍 >

讲述：DS Hunter
时长 23:03 大小 21.12M

你好，我是 DS Hunter。

上一讲，我们谈到了无收益的前端是如何为反爬提供基础保障支持的。但是你应该是知道的，前端没有什么秘密可言。那么，那部分前端不能做的，后端不好做的，我们放在哪里处理呢？

是的，最终还是 BFF 扛下了所有。BFF 天生就是为了这些事情而生的。

在当前的浪潮下，大家似乎将 BFF 与 Node.js 划了等号，事实上并非如此。BFF：Backend For Frontend，服务于前端的后端，仅仅是一种设计思路，只不过 Node.js 在当前环境下是最优解而已。实际上，很多公司都会有一个 API 层来进行服务转发，这一层就相当于 BFF 层，它未必是 Node.js 站点。

这一讲，我们就来看一下 BFF 的主要功能以及实现方式。在学习的过程中，我也希望你能理解 BFF 存在的真正价值。我们先从 BFF 形成的第一步：选型，开始了解吧。

BFF 选型

关于 BFF 选型，理论上我们可以使用任何语言。但是实际上 90% 以上的公司，最终会选择 Node.js。

Node.js：Express

Node.js 可以承载常规 http 请求，并且因为与前端使用相同的编程语言：JavaScript，因此能很方便的让前端人员兼职掉 BFF 工作。这是常规公司选型的理由之一。对于反爬来说，我们有更进一步的选型原因：

1. 我们之前提到的加解密是成对出现的，客户端解密，BFF 加密，出于方便测试的理由，使用 JavaScript 做为 BFF 语言，会十分方便。
2. 反爬虫系统是一个高并发的非关键系统，其余语言需要额外引入支持，Node.js 默认是异步的，天生为高并发而生。
3. JavaScript 代码处理可能涉及 AST 转换， Node.js 有足够多的工具可以支持。

基于这些理由，Node.js 几乎成了唯一解。但是，万事不是绝对的。对于部分公司来说，实在不能选择 Node.js 的时候，也可以被迫使用别的技术栈。

被迫选择其余技术栈的理由千差万别，但通常是历史原因或基建支持不足。但是，现实世界的架构也常常不是最优解。我这里的建议就是：妥协吧，编码不方便，可以通过增加人力来解决，这不是无解的问题。

主要功能：中转以及聚合

BFF 的功能，往往是聚焦于网络请求的中转以及聚合。中转很简单，就是前端服务请求过来，BFF 承接了请求，然后处理后转发给后端。而聚合，则是多个请求被合并，一次性请求后端，降低后端的 QPS。别人用 BFF，往往只是用于中转和聚合，而我们享受了 BFF 的好处，拿来做反爬。那么这里，我们就来讨论一下中转以及聚合部分的机器配置以及相应的反爬操作。

从 BFF 机器配置来看，我们为了承载高并发，就要增加资源来承载。但是增加哪方面的资源？是机器数，还是增加配置？这个取决于场景。

机器配置：网络负载分类

我们经常说高并发高负载，但是实际上业务上存在两种不同的负载方式：一种是请求量多，也就是链接数高。另一种是流量大，或者资源消耗多，也就是每秒的 byte 数多，或者需要的业务逻辑多。这两者的处理方式是不同的。当然也有业务是两者都多，这就没有什么取巧办法了。

针对链接数高，其实可以通过增加机器数来实现。简单的说，比如你使用 1C2G 的机器，顶住了网络流量。这个时候，链接数忽然增加到 8 倍，你是增加至 8 台 1C2G 的机器，还是替换成一台 8C16G 的机器呢？其实两者成本是一致的，但是我们还是会选择 8 台机器，因为更多的网卡意味着更多的负载。

但是是不是永远这样选择呢？也不一定。我们可以看一下第二种情况：流量大、资源消耗多。这种情况下，可能内存大的“一台 8C16G”的机器就是首选了。

当然，事情还有反转的可能，如果系统对实时性要求很高，你可以承受频繁 GC，但是不能承受长时间 GC，那么可能大内存又不是首选了。这个在选型的时候要慎重考虑。

如果非要有一个结论的话，那么其实反爬虫系统大部分是对实时性有要求，所以低配置多机器数通常是首选。但是这并非绝对的，还是要看具体的需求来慎重选择。

具体操作：key 与随机值

有了合适的机器，我们就可以考虑具体操作的问题了。我们从反爬验证的核心问题“key 的生成”开始说起。

key 生成相关

我们在09 讲提到过，key 生成逻辑并不复杂，并且无需存储，因此可以直接在 BFF 生成，不需要动用后端生成再用 BFF 转发。而且加密本身是一个计算密集型的需求，不是 IO 密集型，所以这个对 CPU 的消耗还是比较可观的。

但是，考虑到反爬是一个低 ROI 的项目，成本节约是重中之重。所以，我们来分析一下目前面临的问题：Node.js 天生是为单 CPU 设计的，而 Cluster 本身只是充分利用了多核，并没有什么额外的优化。所以对于 Docker 部署的公司来说，可以把机器分散成多个一核的实例，这样可以充分降低成本。

这里有两个理由。第一，与其用不靠谱的 Cluster 来实现多核优化，还不如用成熟的 Docker 技术来隔离，反正总核数不变。第二，核数少的话，配置 Docker 的人可以使用一些边角料来凑，成本能大大降低。例如公司的 k8s 人员分配了一堆高配机器之后，还剩余一些机器没办法分配出去，由于我们的需求是单核，因此无论剩余什么边角料，都可以使用上，这个成本能低很多。

此外，考虑到 BFF 是一个集群，那么发布过程中，需要考虑 key 兼容的情况。

例如发布了 50% 的机器，那么 50% 机器用了新版的 key 生成与校验逻辑，另外 50% 还使用的是旧版，这里就会存在发布常见的交叉问题。要么发布过程中停掉反爬，要么在新校验逻辑里要兼容上一版。停掉反爬代价会比较大，而且很难进行线上观察测试。

因此我们的验证模块通常会对上一版做兼容。要知道，key 生成并非一个频繁变更的需求，所以必要时可以通过多一次发布来解决这个问题。也就是一次发 key 的生成方式， 一次发 key 的验证方式。

key 验证相关

key 验证方法在09已经讲过了，这里不再赘述。今天我们主要论述的，是针对刚刚提到 key 验证中“多一次发布”的问题产生的相对应的 key 验证问题。

我们先看一下 key 验证的版本问题是怎么产生的。假设我们的 key 生成使用的是 v1 版本，验证使用的也是 v1 版本。那么，我们要将他们都变更为 v2 版本。由于发布途中的 n 台机器，只有 1 台发布完成了，那么如果用户请求 key 的时候访问了 v2 版本生成的 key，然后取 v1 版本去验证 key，毫无悬念，这个验证是不可能通过的。那么这个时候我们就需要进行线上版本兼容处理。

首先，发布的新版本应该支持所有前置版本的 key 生成以及验证逻辑，并且支持配置。例如，我们刚刚发布途中，1 台机器支持 v2 的 key 生成与验证，n-1 台支持 v1 的生成与验

证，那么我们的配置应该是：生成 key 配置使用 v1，验证 key 配置也使用 v1。全部发布完毕之前，实际上线上一直使用的是 v1 版本。全部发布完毕之后，我们可以将一台机器的 key 生成调整为 v2，而验证 key 的配置设置为同时支持 v1 与 v2。

这样，线上部分用户使用 v2 版本的 key，大部分用户使用 v1 的 key。慢慢地，就可以逐渐灰度切换为 v2 版本 key 了。然后全部切换完毕，v1 版本 key 消失，此时验证 key 版本设置为 v2。我们就可以进行灰度的切换了。

整个过程会缓慢执行，**同时，要时刻监控我们前面提到的误伤统计报表。**这个报表是实时的，所以可以用来做发布监控，这也是他存在的最大意义。但是这里要注意一个细节：我们误伤检测的理论，是从价格页面带 Cookie 到订单页面，检测 Cookie，假设没有 Cookie，就认为没问题。这个在平时没有问题，**但是在发布时要考虑下用户操作延时这点。**

简单的说，用户从价格页到订单页是有一个延迟的，任何一个用户访问了价格，都要通过几分钟各种比较，才会真正下单，根据经验一般是 5 到 15 分钟不等（不过这个并非定值，每个公司应该不一样，可以在 Cookie 上加上时间戳，自己线上测试下符合自己业务的时间段），因此灰度变更的时间间隔不应该小于这个时间差，否则可能导致命名误伤了用户，但是误伤检测还没到，发布的时候误以为策略是安全的，最终因为变更过快，引发生产事故。

随机值处理

从刚刚的发布相关论述中，你会发现，我们通过集群实现了一个随机。实际上，由于集群本身是随机访问的，这里相当于有了一个天生的随机模块。所以我们可以使用集群本身直接做随机。这句话你可能不太理解，没关系，我们还是通过一个例子来讲解具体的操作方式。

举个例子，我们在09 讲中提到过的随机放过。假设你想随机放过 50% 的爬虫，那么我们并不需要编写随机模块，对每次请求进行 roll 操作。实际上，通过集群进来的访问已经随机过了，他们被随机并且平均分配到了集群的每一台机器上。你只需要针对 50% 的机器，配置上“对于任何验证 key 的请求，不进行验证，直接判定为 key 正确”，就可以实现 50% 的随机放过了。

这样不仅降低了代码复杂度，避免出现随机不准的问题，还提升了效率。属于集群使用的小技巧了。当然，这种小技巧除了反爬，似乎也没有别的业务用得到了。

这里可以给你补充一点，那就是如果你需要比较特殊的随机值，例如你有 10 台机器，你需要的概率却是 55%，这个需要变更 5.5 台机器，很难做到，你可以通过变更 Nginx 的 weight 来实现，详情可以参阅 Nginx 的官方文档。不过这样会导致负载不均衡，因此建议是用工整的随机值。如果不工整，可以向上向下取值到一个合适的值。

实现方式：自定义 Node.js 的 engine

实际上，我们可以通过定制 Express 的 Engine，创造一个新的扩展名来避免大量的切片，直接实现 BFF 中转以及聚合的复杂操作。

Express 的 Engine 定制方法

我们都知道，Express 支持自定义 Engine。默认情况下，框架提供了一个非常好用的 Engine。但是问题就是，他太好用了，一点也不混乱。而我们做反爬，就是要做混乱，让爬虫方摸不到头脑。因此，定制 Engine 是一个势在必行的事情。

关于 Express 如何编写自己的 Engine，可以查阅官方文档。我们这里主要讨论，哪些方面我们要变更。

engine 混淆

Engine 的混淆主要包括：变量名混淆，eval 博弈，AST 转换以及虚拟机。

这里，我主要用变量名混淆给你举例。至于其余的混淆方式大同小异，都是定制 Engine，只是复杂度会提升不少。所以，我们先用稍简单些的变量名混淆理解基本原理。

我们假设，原来是如下的代码：

复制代码

```
var a = 3;
var b = 4;
console.log(a-b);
```

但是，我们希望给爬虫方展示的是这样的代码：

复制代码

```
var $asbasdfewf = 3;
var $asbasdfevf = 4;
console.log($asbasdfewf - $asbasdfevf);
```

明显可以看得出，虽然只有三行的 demo，但是下文的阅读难度比上文复杂了不少。

但是我们不能容忍自己在写代码的时候，书写下面的代码。恶心的代码是给爬虫看的，不是给自己看的。那么我们如何处理呢？

我们可以定义一个 engine。假设定制完毕后，我们的 engine 扩展名是 as：代表 anti-spider， 那么也许你的 index.as 文件可能看起来类似这样：

复制代码

```
var $a = 3;
var $b = 4;
console.log($a-$b);
```

然后，你的 engine 可能定制类似这样的代码：

复制代码

```
// 遍历模版， 获取所有的变量名。
// 可能是正则， 可能是AST， 看你喜欢， 以及实现难度
// 设置到variables里。
var variables = getAllVariables();
// 根据所有的变量， 创建一个字典， 实现旧变量名与新变量名的一对一mapping。
// 新变量名可以尽可能长， 并尽可能视觉上相似。 例如混用v和w， 1和l之类的。
// 为了避免数字开头， 可以使用通用的开头， 例如$
var dict = createVariableDictionary(variables);
// 批量替换查找到的所有变量名。
// 老规矩， 正则或者AST
replaceAll(page, dict);

```

这样，我们就可以看到干净整洁的代码，而那些恶心的代码，就让爬虫一点点看去吧。

如果上面的代码看着迷惑，没关系，这说明你缺乏的是 express 定制 engine 的知识，而不是缺乏反爬虫知识。你先去跟着官网 demo 定制一个简单的 engine 看下流程，再回来看就能明白我们要干什么了。

其余的都是用类似的思路去做即可。定制 engine 虽然是个很非主流的功能，甚至很多人都觉得 Express 支持这个东西简直匪夷所思。但是对于反爬虫来说，这简直是神器。

engine 效率

定制 engine 最需要注意的就是效率。

我们为了方便，很可能大量使用正则表达式对整个模版进行字符串扫描。这个复杂度至少是 O（n）的。而我们又需要频繁扫描，加上不断的局部调整，可能导致 CPU 过高。众所周知，JavaScript 的字符串操作性能堪忧。

因此，这里可以进行一些必要的提速操作。在迫不得已的情况下，甚至可以用其他语言来实现 char 数组操作，就像编译原理课上的作业一样，尽可能在较少的全字符串扫描中完成任务。这可能会很耗开发资源，但是绝对是值得的。还记得咱们在09 讲中提到过的几乎无解的计算力进攻嘛？这样的改变可以在一定程度上缓解这样几近疯狂的进攻。

也就是说，你消耗的资源不能过高，否则将产生资源的不对等，爬虫消耗的资源远远低于你，你就会陷入被动。而如果你提升了效率，消耗的资源并不高，那么你就可以相对放心很多。

engine 切片

切片主要是针对一些浏览器特征检测的。

我们假设你要检测 location.href 是否存在某个字符串（你的 URL 包含一个关键字，那么这里 location.href 就应该包含这个关键字），那么一定不要直接写：如果不包含就如何如何。这样被调试出来的概率太高了，对方只要查找一下 if，就知道所有的判定在哪里了。那么这个代码应该怎么写呢？

如果你的某一步加密操作是 +3，那么你的代码可以这样写：

复制代码

```
1 var offset = 3*(+!!(location.href.indexOf(keyword)+1)
```

> 代码逻辑大概是：
>
> 如果存在 keyword，那么 +1 之后将得到一个大于 0 的数字。这个数字取反为 false，再取反为 true，再加个加号就是 1 了。然后用 3 来乘这个数字，就得到了正常的 offset。
>
> 如果没有找到，那么它是 -1，-1+1 得到 0，两次取反为 false，再用加号转为 Number，是 0，3*0 还是 0，这个 offset 就是 0 了。
>
> 如果 offset 错了，那么会导致什么结果呢？会导致 key 解密错误，最终验证的时候，验证失败。
>
> 但是，整个过程没有任何一个 if，是不能直接查找搜索得到结果的。

那么这个代码我们是直接写到解密里面吗？不是的，我们代码里会放 3*1，1 就是一个切片，会被替换成这段 js 判定，在不同条件下运行出不同的结果。

那么替换的动作，就要由 Engine 来执行了。同时，这个替换是随机的，也就是不同的切片与不同的切片代码来互相替换，实现代码的随机性。我们知道，随机出现的代码是最影响调试的。

当然，这个切片只是个思路，核心在于消除 if 判定，而不是全篇都是这样操作，要想各种办法来变通。如果千篇一律，对方只要搜索"+!!"这种关键字就可以找到你的切片了。万幸，js 有大量的特性（bad parts），可以供我们试用。

Engine 下的成对加解密

我们在10 讲中提到了解密方法是逆排序的，那么这里就可以顺便看下做法了。简单的说，既然是逆序，那就是后进先出，也就是一个简单的栈操作。思路确定后，那么问题就简单了。

大体来说，代码如下：

```
// 首先我们从题库roll出来几各加解密的pair
// 然后把他们临时保存一下
var methods = rollMethods();
// 我们设定一个
var methodStack = [];
// 这里要对methods进行遍历
// 针对key进行加密， 每加密一次， 就在methodStack进行一次push操作
// methods的每一个item应该是一个加解密pair， 加密方法当场用掉，
// 解密方法是一个字符串， 留在栈里备用
methods.forEach...

while(methodStack.length !== 0){
  // 这里则需要不断的popmethodStack了。
  // 然后推给engine， 让engine按顺序拼接解密方法
  // engine拿到解密方法， 进行字符串拼接， 切片注入
  // 以及其余的混淆， 然后加入通用模版， 得到浏览器端最终运行的js代码
}
```

这样，我们的解密 method 无需与加密 method 进行 mapping，因为加密方法当场用掉了，解密方法拼接到 js 中了，两者都消费完了。所以不需要再 mapping 了。浏览器端解密完毕，就直接得到 key。如果得不到，那就是解密失败，会被验证模块拦截掉。

附加价值：BFF 集成与解耦

目前为止，BFF 的反爬任务就基本上完成了。但是我们创建 BFF 不仅仅是替后端做脏活累活的，还需要有更多的价值。我们先从反爬的初心开始讲起。

后端保护

反爬虫的初心其实是防止后端收到大流量冲击。现在看来这反倒成了最低级别的需求了。不过我们还是不忘初心，看下 BFF 是如何保护后端的。

我们知道，真正能冲击后端的爬虫，其实反倒不是那些竞对爬虫，而是毕业生爬虫，他们在毕业季拼命爬取数据， 用来做毕业设计，写论文。而且他们没什么技巧，就是死命干。所以简单的规则封杀就可以操作掉。

这类封杀， 如果在 SLB 层不方便做，在 BFF 层可以轻易实现，本质就是一个规则过滤——而我们已经有规则引擎了。因此，触发规则之后，直接设置 state 为结束，流量就会被忽略掉。后端压力也就得到了降低。 而 BFF 本身也不用担心，因为在选型时选择了 Node.js，应对高并发是天赋，受到的冲击极为有限。

当然，有些公司的 BFF，我们提到过不是 Node.js……因为一些特殊原因……这种还是死磕 SLB 去做规则吧。

前端熔断

除了保护后端，BFF 也可以轻易做到前端的熔断。

我们的反爬验证，无非就是 key 的生成和 key 的验证，那么最彻底的熔断当然是不生成 key 也不验证 key。但是有时候线上问题紧急，多个开关操作起来，会互相干扰。

所以，有时候也可以只操作验证 key 开关，简单的说，就是无论 key 错得多离谱，我们的规则引擎都默认设置为通过。

我们前面提到过随机放过的操作，可以在集群直接操作。实际上，你把概率设置为 100%，所有集群都放过，那么也就等于熔断了。快速熔断，关键时刻可以救命，一定要做好。

超时处理

超时处理与熔断其实差不多，那么具体的实现逻辑是怎样的呢？

首先，生成 key 的时候，不能让用户等太久。如果超时了，那么就可以迅速生成一个指定的 key，让他先走掉。固定 key 很难通过验证的，因此这个报警一定要快，因为误伤可能已经产生了。由于我们的误伤操作通常是操作价格，并不是拦截，因此会给予你一定的时间来处理。

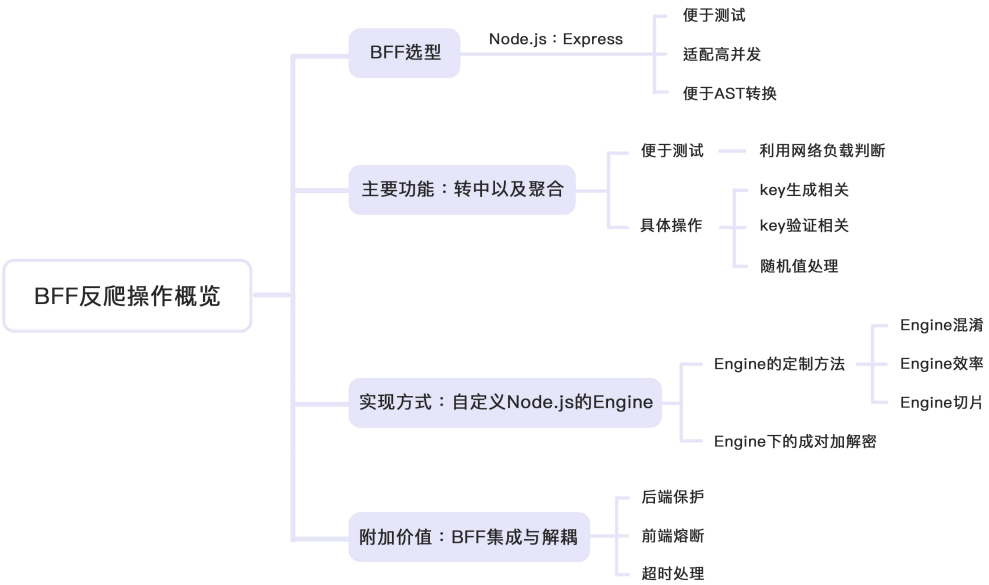
此外，验证 key 也需要做超时处理。假设一个 key 验证较长时间还没验证完毕，那么不管哪里逻辑有问题，规则引擎直接给出“验证通过”的结果即可。一定要记住：反爬是一个非关键的业务，抓不到爬虫不可耻，引发生产问题才可耻。

此外还可以设置一些自动熔断的规则。注意，如果这些规则被爬虫抓到，他可能有意触发。所以，还是那句话：具体问题，具体分析。这个需要权衡以及尽可能隐蔽，不要被抓到规律。

小结

好了，最后我来给你总结一下。

今天这一讲，我们了解了 BFF 在反爬虫动作中的一系列问题。从选型开始，到实现反爬虫的详细动作，以及实现反爬虫功能的详细方式都做了详细的讲解。最后，我们也补充了 BFF 在集成和解耦上面的附加价值。详细的内容我也给你准备了一张脑图，你可以对照着来复习。



但是，这里我更想和你强调几个这一讲的重点，也是 BFF 在反爬虫这件事上最关键的几点：

1. **自定义 Engine：**BFF 通过自定义 Engine，来实现 key 的生成以及验证；
2. **通过集群设置随机值：**使用 weight 定义机器权重，实现加权分流，并能过控制机器开关实现随机；
3. **BFF 小技巧：**通过自定义随机值来设置熔断、通过设置不验证实现熔断、通过超时处理来提升用户体验。

那么，到这里为止，我们的理论课就告一段落。下一讲，我会带你从头到尾走一遍反爬虫的过程，认真观察这个战场的每个角落。

思考题

又到了愉快的思考题时间。 老规矩，你可以任选一个问题和我讨论。

1. 你们的公司出于各种原因，BFF 没有选择 Node.js 技术栈。那么，你是硬着头皮在这种技术栈做反爬呢，还是想办法让 BFF 转 Node.js 呢？如果你认为不可能成功，那么理由是什么呢？
2. Express 的官方自定义 Engine 的 demo 里面，会频繁读文件。虽然 Node.js 是异步的，但是也顶不住没完没了的读硬盘啊。如何提升性能呢？
3. 集群实现随机，你精心配置，终于实现了 79.4%。然后机器扩容，随机值又变了。那么，如何处理这个问题呢？

期待你在评论区的分享，我会及时回复你。反爬无定式，我们一起探索。

极客时间

左右纠结难上线，前后为难宜中庸。

反爬虫兵法演绎 20 讲

DS Hunter / 反爬虫专家，前某公司研发总监

识别二维码
免费试读

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 0 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 答疑课堂 | 爬虫反爬虫调试对抗技巧以及虚拟机详解

下一篇 13 | 反爬虫基础理论：这是唯一一节严肃的纯理论课

精选留言 (2)

写留言

peter

2022-02-25

请教老师几个问题：
Q1： AST转换是指什么？
Q2： 机器核数一般10核，那么2倍是20个连接。但tomcat需要创建500个连接？
Q3： Node.js 天生是为单 CPU 设计的，而 Cluster 本身只是充分利用了多核
Q4：Node.js天生高并发，它的并发处理能力多大？比Nginx还强吗？...

展开

作者回复: 好多，手机不方便回一部分吧，node不见得能超过nginx，但是node写逻辑是写js，这一点nginx比不了。实际上很多node前面还放一个nginx。cluster用了多核，不过并不会有额外收益，所以宁可选择用docker分。ast 就是语法转换，webpack有专门的文档介绍可以参考，能改代码结构，我们用来做加密。

leslie

2022-02-25

均衡看似简单，实则如西天取经。
集群随机的问题聊聊自己的薄见，不知老师是和观点？
集群部分随机，如同兵法～实实虚虚，实则虚之，虚则实之；不过这又给均衡带来了难题-哪里均衡？

作者回复: 都是踩钢丝行为，如履薄冰